ACADEMIC PRESS

Available at www.ComputerScienceWeb.com POWERED BY SCIENCE DIRECT®

Journal of Computer and System Sciences 66 (2003) 763–774

JOURNAL OF COMPUTER AND SYSTEM SCIENCES

http://www.elsevier.com/locate/jcss

Two-dimensional substring indexing

Paolo Ferragina,[a,1] Nick Koudas,[b] S. Muthukrishnan,[b,*] and Divesh Srivastava[b]

[a] *Dipartimento di Informatica, University of Pisa, Corso Italia 40, 56125 Pisa, Italy*

[b] *AT&T Labs-Research, 180 Park Avenue, Building 103, Florham Park, NJ 07932, USA*

Received 4 September 2001; revised 20 May 2002

Abstract

As databases have expanded in scope to storing string data (XML documents, product catalogs), it has become increasingly important to search databases based on matching substrings, often on multiple, correlated dimensions. While string B-trees are I/O optimal in one dimension, no index structure with non-trivial query bounds is known for two-dimensional substring indexing. In this paper, we present a technique for two-dimensional substring indexing based on a reduction to the geometric problem of identifying common colors in two ranges containing colored points. We develop an I/O efficient algorithm for solving the common colors problem, and use it to obtain an I/O efficient (poly-logarithmic query time) algorithm for the two-dimensional substring indexing problem. Our techniques result in a family of secondary memory index structures that trade space for time, with no loss of accuracy.

1. Introduction

Conventional databases, and the index structures defined on these databases, have been designed for business data, e.g., customer and facilities information. Over the years, and particularly now, on account of the rapid growth of the Internet and XML, there has been a growing need to manage, and index, string data. Several index structures for string data have been proposed, beginning with the classical work in [3,13,14] to the more recent secondary memory structures in [5,10]. Much of this work has dealt with indexing a single string attribute.

*Corresponding author.

E-mail addresses: ferragina@di.unipi.it (P. Ferragina), koudas@research.att.com (N. Koudas), muthu@research.att.com (S. Muthukrishnan), divesh@research.att.com (D. Srivastava).

[1] Work partially supported by Italian MIUR projects on "Technologies and services for enhanced content delivery" and "Piattaforma distribuita ad alte prestazioni".

0022-0000/03/$ - see front matter © 2003 Elsevier Science (USA). All rights reserved.
doi:10.1016/S0022-0000(03)00028-X

Multi-dimensional matching of string data is important in a variety of contexts. For example, in XML databases, many attributes and elements tend to be string-valued (of type PCDATA). Almost any sufficiently complex XML query involves selections specified on multiple such attributes and elements. In many contexts where strings are used, supporting partial match queries, such as prefix and substring matching, which are more common and more useful than exact matching, is crucial. In an XML bibliographic database, for example, one might ask for articles whose title contains "string" as a substring, and whose author's name contains "muthu" as a substring. Even in relational data warehouses, one needs multi-dimensional substring querying. For example, one can imagine asking queries that select on a prefix match of the supplier name and a substring match of the product name. While there is a rich history of research in multi-dimensional index structures for numeric data (see, for example, [6,15]), only recently has there been research on multi-dimensional string indexing. In particular, Jagadish et al. [10] described a general technique for adapting arbitrary numeric multi-dimensional index structures (such as R-trees) to support wild-card indexing of string data. Most commercial search engines rely on the one-dimensional (1-d) approach described in the second option below, and use an inverted index on each dimension.

When matching the two-dimensional (2-d) substring query (σ_1, σ_2), which asks for all database string pairs $\alpha_i = (\alpha_{i,1}, \alpha_{i,2})$, such that σ_1 is a substring of $\alpha_{i,1}$ and σ_2 is a substring of $\alpha_{i,2}$, one has two options currently:

- First, use a 2-d string index structure, such as a 2-d string R-tree [10]. With k string pairs in the database, of average length l, if one wishes to ask 2-d substring queries, the string R-tree must store $O(k \cdot l^2)$ "points", and queries correspond to 2-d rectangles.[2] While the string R-tree is expected to perform well in practice, no non-trivial (i.e., sublinear in the number of points in the R-tree) worst-case cost is known in this case.
- The second option is to use two distinct 1-d string index structures, such as the string B-tree [5], one in each dimension, and then merge the partial results to get the desired multi-dimensional query result. To support substring queries, each string B-tree index must store $O(k \cdot l)$ "points", and queries correspond to 1-d ranges.[3] In this case, string B-trees have a known worst-case query cost of $O(\log_B(k \cdot l) + |\frac{\mathsf{occ}}{B}|)$, where $|\mathsf{occ}|$ is the number of matching occurrences of a substring in that dimension, and B is the blocking factor; this matches the bound known for numeric B-trees. However, when the query strings in each dimension are not very selective, but their combination is very selective, the sizes of the partial results may be arbitrarily larger than the size of the final query result. Thus, the use of 1-d string indices is quite inefficient as well for 2-d substring matching.

1.1. Contributions and outline

In this paper, we provide the *first* non-trivial worst-case query cost for the 2-d substring matching problem, without using asymptotically more space than the string R-tree approach [10].

[2] If only prefix or range queries are asked, the 2-d string R-tree needs to store only $O(k)$ points, independent of l.
[3] If only prefix or range queries are asked, the string B-tree needs to store only $O(k)$ points, independent of l.

This paper makes the following technical contributions:

- We present a reduction of our 2-d substring indexing problem to the geometric problem of identifying common colors in two numeric ranges containing colored points.
- For the common colors problem, we develop a family of 2-d secondary memory index structures that trade space for time, with no loss of accuracy. With N colored points, members of the family contain $O(N^{1+\varepsilon})$ points, $0 \leqslant \varepsilon \leqslant 1$, on an $N \times N$ grid.[4]

  There are two results of interest. First, for the family member that contains $O(N^2)$ points, the worst-case query time has poly-logarithmic I/O complexity. Second, there is a "sweet spot" in this family that contains $O(N^{4/3})$ points, while maintaining an expected-case poly-logarithmic I/O complexity.
- We show that when reducing from the 2-d substring indexing problem, on a database containing k string pairs of average length l, the 2-d index structure for the common colors problem contains $O(k \cdot l^2)$ points, and query time is poly-logarithmic in the worst case.
- We show that the two options of using (i) a 2-d index structure (with $O(k \cdot l^2)$ points, and a poly-logarithmic access cost) and (ii) two 1-d index structures (each with $O(k \cdot l)$ points and an $O(k)$ access cost) are simply two ends of a spectrum.

  We illustrate that it is possible to combine the use of these two access structures in a way that stores $O(k \cdot l^{1+\varepsilon})$ points, $0 \leqslant \varepsilon \leqslant 1$, using poly-logarithmic access cost for some 2-d substring queries, and using $O(k)$ access cost for others. This allows the user to choose a point in this spectrum, depending on available space and time constraints.

Our results provide the *first* non-trivial bounds known for the 2-d substring indexing problem, without taking asymptotically more space than the 2-d string R-tree. All our results are in fact presented in the standard external memory model and capture the complexity of disk accesses with large datasets.

We present related work in Section 1.2, before formally defining our problems in Section 2. We present our algorithms for the geometric problems in Section 3, and then use these to solve the 2-d substring indexing problem in Section 4. Finally, in Section 5, we present a divide-and-conquer technique resulting in a family of algorithms that trade space for time, without loss of accuracy.

1.2. Related work

Several variations of range queries (e.g., colored, weighted, etc.) have been previously studied in the main memory model [1,8]. Efficient retrieval of substrings in one dimension has been studied by Matias et al. [12], for the main memory case.

The string B-tree was proposed in [5]. In this paper, the authors introduce the notion of an index structure designed for unbounded length strings. String B-trees can answer substring queries in one dimension by indexing all the suffixes of the strings in the string collection. Jagadish et al. [10] extended the ideas behind string B-trees to deal with multiple dimensions, realizing their framework in the context of 2-d R-trees. They provided the first index structure capable of

[4] See Theorem 5.1 for the precise trade-off.

indexing 2-d strings of unbounded length in secondary storage. String R-trees can answer 2-d substring queries by indexing all suffix pairs of each string pair in the database.

Multi-dimensional indexing has a long history in database research. There are numerous proposals for indexing techniques in multiple dimensions. A very good recent survey can be found in [6]. Most indexing techniques can be divided into two major categories: those employing *space decomposition* and those relying on *entity grouping*. Space decomposition techniques apply a hierarchical, canonical or biased, subdivision of the indexed space into a number of non-overlapping regions, up to a point where indexed entities can be mapped into pages. Representatives in this category include the classical Quad trees [15]. Entity grouping techniques organize the indexed space into a hierarchy of (possibly overlapping) regions, and map regions into disk pages in the last level. The R-tree family [4,9,16] of indexing structures in based on this principle.

There exist several structures for indexing strings in one dimension. The Prefix B-tree [3] is a classic structure, capable of dynamically handling 1-d variable length strings. String B-trees [5] provide better performance guarantees for variable length strings than Prefix B-trees, however. Suffix arrays [11] and PAT-arrays [7] allow for fast searches on strings but they are difficult to update in secondary storage. Suffix trees [13] are another classical index for strings; they have an unbalanced tree topology, which makes their dynamic maintenance in secondary storage difficult.

2. Definitions

In this paper, we use greek letters α, σ (with and without subscripts) to refer to strings and string pairs. We now formally define our 2-d substring indexing problem.

Definition 2.1 (2-d Substring indexing problem). Let D be a database consisting of a collection of string pairs $\alpha_i = (\alpha_{i,1}, \alpha_{i,2})$, $1 \leqslant i \leqslant k$, which may be preprocessed.

Given a query string pair (σ_1, σ_2), the 2-*d substring indexing problem* is to identify all string pairs $\alpha_i \in D$, such that σ_1 is a substring of $\alpha_{i,1}$ and σ_2 is a substring of $\alpha_{i,2}$. □

We use $\ell_{i,j}$ to denote the length of string $\alpha_{i,j}$, and use M to denote $\sum_i (\ell_{i,1} + \ell_{i,2})$.

For example, we may have a database of names and phone numbers of people and a query may be to retrieve all people with `bill` in their name and `202` in their phone number, (`bill jones, 202-555-1234`) being an example.

Our overall approach is to reduce the 2-d substring indexing problem to geometric problems described below.

Definition 2.2 (Colored range query). The *colored range query* (CRQ) problem is as follows. We are given an array $A[1 \ldots N]$ of colors drawn from $1, \ldots, C$. We want to preprocess this array so that the following query can be answered efficiently: Given a range $[a, b]$, list the *distinct* colors that occur in this range.

In Fig. 1, the CRQ problem on interval I_1 results in colors $\{R, G, B\}$, whereas the CRQ problem on interval I_2 results in colors $\{Y, R, G\}$, independent of the number of occurrences of these colors in the respective ranges.

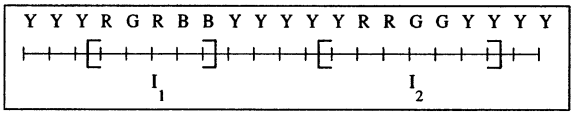

Fig. 1. CRQ and CCQ problems.

Definition 2.3 (Common colors query). The *common colors query* (CCQ) problem is as follows. We are given an array $A[1\ldots N]$ of colors drawn from $1, \ldots, C$. We want to preprocess this array so that the following query can be answered efficiently: Given two non-overlapping intervals I_1 and I_2 in $[1, N]$, list the *distinct* colors that occur in *both* intervals I_1 and I_2.

In Fig. 1, the CCQ problem for intervals I_1 and I_2 results in colors $\{R, G\}$.

In this paper, we will work in the standard external memory model [17], where the input resides on a disk and all accesses to it are in pages of size B, and m is the ratio of the main memory size to the disk page size. As is usual, we measure the performance in terms of the number of disk pages accessed; in addition, we will focus on the number of disk pages used for the solution.

3. Solving the CRQ and CCQ geometric problems

In this section, we provide I/O efficient algorithms for the CRQ and the CCQ geometric problems.

3.1. The CRQ geometric problem

First, we focus on the CRQ problem. We are given an array $A[1\ldots N]$ of colors drawn from $1, \ldots, C$, and want to identify the distinct colors that occur in some subrange I_1 of $[1\ldots N]$. The naive solution of using an indexing structure such as a B-tree to solve this problem takes $O(\log_B(N) + \frac{|I_1|}{B}\log_B(\frac{|I_1|}{B}))$ disk accesses. Since $|I_1|$ may be significantly larger than the number of distinct colors occurring in the interval, this (naive) solution is not always efficient. An I/O efficient algorithm is given below.

Theorem 3.1. *There exists a data structure to solve the CRQ problem using $O(N/B)$ disk pages of size B in which each query takes time $O(\log_B(N) + \frac{\texttt{output}}{B})$, where* `output` *is the number of colors in the queried interval.*

Proof. We generate points $(i, p(i))$ of color c where $A[i] = c$ and $p(i)$ is the largest index smaller than i such that $A[p(i)] = c$; when there is no $p(i)$, we set it to $-\infty$. The p function therefore gives the predecessor of i of the same color as itself for any i. This set of points is denoted S.

We construct a set S_c formed by points $(i, p(i))$ of color c (i.e. $A[i] = c$), and define $S = \bigcup_c S_c$. In [8], it was shown that the query interval $[a, b]$ contains at least one point colored c if and only if the 3-sided rectangle $[a, b] \times (-\infty, a)$ contains a point of S_c. Moreover, if that point of S_c does exist, then it is unique. The main idea in the proof of [8] is that among all the points colored c in

$[a,b]$ only the *leftmost one* gives a point of S_c which lies in $[a,b]\times(-\infty,a)$. This point constitutes the *witness* of the existence of a point colored c in $[a,b]$.

As a result the CRQ problem can be solved by listing the colors of the points of S lying into $[a,b]\times(-\infty,a)$; we will have just one witness per color. This 3-sided rectangle query can be solved by using the data structure in [2] that occupies $O(N/B)$ disk pages and requires $O(\log_B(N)+\frac{\mathtt{output}}{B})$ disk page accesses. □

We make an important observation on the result above: it holds even in the case when N is the number of colored points in A whereas A may contain other non-colored points. That is, the actual size of A does not play any role provided that it is representable in one machine word. This observation will be used below since we will deal with sparse arrays shortly.

3.2. The CCQ geometric problem

We next focus on the CCQ problem. We are given an array $A[1\ldots N]$ of colors drawn from $1,\ldots,C$. We want to preprocess this array to efficiently identify the distinct colors that occur in both the non-overlapping intervals I_1 and I_2 in $[1,N]$. We reformulate this problem as follows.

We construct a matrix AA in which $AA[i,j]=c$ if and only if $A[i]=A[j]=c$. Thus, AA is a (possibly sparse) $[1,N]\times[1,N]$ matrix. Any query to AA will be a rectangle, that is, $[a_1,b_1]\times[a_2,b_2]$ and it returns the distinct colors in the rectangle. A query for the CCQ problem on array A with input intervals I_1 and I_2 is the same as a query to matrix AA with input $I_1\times I_2$; hence, we will focus on the rectangle query and matrix AA henceforth. Our algorithm proceeds as follows.

3.2.1. Preprocessing

We consider the N columns of matrix AA and construct the *χ-adic grouping*. That is, for a fixed χ and for all integers κ and i, we consider *metacolumns* by concatenating columns $\kappa\chi^i+1,\kappa\chi^i+2,\ldots,\kappa\chi^i+\chi^i$. For example, the set of all metacolumns with $i=0$ is $\{1,2,\ldots,N\}$; with $i=1$ and $\chi=2$, we get $\{(12),(34),\ldots,((N-1)N)\}$, and so forth. Clearly, the maximum possible value of i is $O(\log_\chi(N))$. Fig. 2 illustrates the metacolumns that arise for a χ-adic grouping, with $\chi=3$.

Next, we "linearize" the metacolumns row-wise. That is, for each metacolumn $\kappa\chi^i+1\cdots\kappa\chi^i+\chi^i$, we consider a 1-d array $AAA_{\kappa,i}$ such that $AAA_{\kappa,i}[(j-1)\chi^i+l]=c$ if and only if $AA[j,\kappa\chi^i+l]=c$. We preprocess each of the $AAA_{\kappa,i}$'s as described above for solving the CRQ problem.

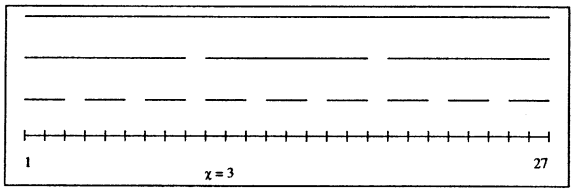

Fig. 2. χ-adic grouping.

3.2.2. Query processing

Given a query rectangle $[a_1, b_1] \times [a_2, b_2]$ on the matrix AA, we decompose $[a_1, b_1]$ into its *maximal*, *disjoint* χ-adic components $(\kappa_1, i_1), (\kappa_2, i_2), \ldots, (\kappa_l, i_l)$, that is, $[a, b] = [(\kappa_1\chi^{i_1} + 1)\cdots(\kappa_1 + 1)\chi^{i_1}(\kappa_2\chi^{i_2} + 1)\cdots(\kappa_2 + 1)\chi^{i_2+1}\cdots(\kappa_l\chi^{i_l} + 1)\cdots(\kappa_l + 1)\chi^{i_l}]$. Note that the i's need not be distinct because up to $2 * \chi - 2$ components may be present in each level, but $(\kappa_j\chi^{i_j} + 1)\cdots(\kappa_j + 1)\chi^{i_j}$'s are distinct for different j's. The maximal decomposition, i.e. one in which no combination of the χ-adic components can be replaced by one of larger i, can be easily found greedily: starting from a_1 and walking right to the closest χ-adic endpoint one after the other, always taking the largest possible power of i. Fig. 3 shows the χ-adic decomposition of interval [7,24], where the check-marked intervals are its maximal, disjoint χ-adic components. It is easy to see that this decomposition is unique and the number of χ-adic components of interval $[a_1, b_1]$ is given by $l = O(\chi \log_\chi(N))$.

We solve the CCQ problem on $[a_1, b_1] \times [a_2, b_2]$ by solving the CCQ problem on each of the χ-adic components (κ_j, i_j) above with interval $[a_2, b_2]$ on the y-axis for the entire width. It is easy to observe that this is precisely the CRQ problem on the linearized array AAA_{κ_j, i_j} corresponding to this metacolumn $(\kappa_j\chi^{i_j} + 1)\cdots(\kappa_j + 1)\chi^{i_j}$. The union of all the outputs for the different χ-adic components gives the output for the CCQ problem. Hence, it suffices to solve the $O(\chi \log_\chi(N))$ CRQ problems on the appropriate metacolumns. This gives us the following result:

Theorem 3.2. *Let T be the total number of non-zero entries of matrix AA. The CCQ problem can be solved using $O(\frac{T+N}{B}\log_\chi(N))$ disk pages, for any value of the parameter χ. Any query $[a_1, b_1] \times [a_2, b_2]$ takes*

$$O\left(\chi \log_\chi(N)\left[\log_B(N) + \frac{\mathtt{output}}{B}\log_m\left(\frac{\chi \log_\chi(N)\mathtt{output}}{B}\right)\right]\right)$$

page accesses in the worst case, where m is the ratio of the main memory size to the disk page size.

Proof. The space bound follows by observing that the array AAA can be represented by just keeping the non-empty metacolumns and their indices. Every point is contained in $O(\log_\chi(N))$ metacolumns, so that the overall space needed to represent all of them is $O(\frac{T}{B}\log_\chi(N))$ disk pages. In addition, $O(\frac{N}{B}\log_\chi(N))$ disk pages are needed to keep track of the indices of the non-empty metacolumns. By the observation following Theorem 3.1, we know that the data structure solving the CRQ problem requires a space which is proportional to the number of indexed points.

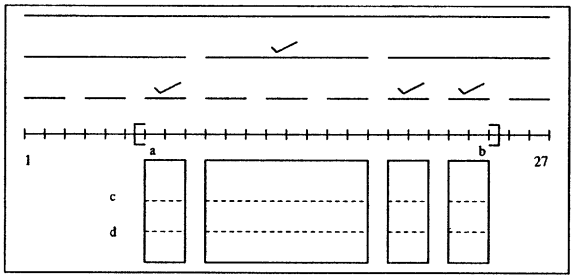

Fig. 3. CCQ query processing.

Consequently, the space bound for the CCQ problem is given by the sum of the previous two quantities.

We have to solve $O(\chi \log_\chi(N))$ CRQ problems on the appropriate metacolumns and then merge all of their results (i.e. lists of colors). Since the output of each CRQ problem is included in the final list of results, we can bound the output size of each query by the value $\texttt{output}$. So the total size of these lists is $q = O(\chi \log_\chi(N)\texttt{output})$.

From Theorem 3.1, we know that all of the above CRQ problems can be solved in $O(\chi \log_\chi(N)[\log_B(N) + \frac{\texttt{output}}{B}])$ disk page accesses. The final list of colors lying in the rectangle query $[a_1, b_1] \times [a_2, b_2]$ can be obtained by removing the multiple occurrences of each color from the previous output lists, via an external sorting routine [17]. This sorting process takes $O(\frac{q}{B}\log_m \frac{q}{B}) = O(\frac{\chi \log_\chi(N)\texttt{output}}{B}\log_m(\frac{\chi \log_\chi(N)\texttt{output}}{B}))$ disk page accesses. Summing up we get the bound stated in the theorem. □

We must choose the parameter χ to be large to optimize the space used, but keep it small in order to optimize the query performance.

4. Solving the substring indexing problems

We solve the 2-d substring indexing problem by reducing it to the CCQ problem as follows.

We build one string B-tree ST_1 on the set of all suffixes of the strings $\alpha_{i,1}$, and one string B-tree ST_2 on the set of all suffixes of the strings $\alpha_{i,2}$. Each (disk page) leaf of ST_q contains a set of string suffixes, ordered lexicographically. We label each suffix $\alpha_{i,q}[j \cdots |\alpha_{i,q}|]$ with the tuple-id i. Next, we generate an array A_1 of tuple-ids by reading the suffixes in the (disk page) leaves of ST_1 from left to right; let A_2 be the array generated the same way from ST_2. We obtain A by concatenating A_2 to A_1; A has total size M since we have one entry per string suffix. This gives us the input to the CCQ problem. Each tuple-id is a distinct color.

Query processing works as follows. Given a query string pair $\sigma = (\sigma_1, \sigma_2)$, we perform a range search for the substring σ_1 in ST_1 and for the substring σ_2 in ST_2. The search for σ_1 returns the contiguous segment of string suffixes having σ_1 as a prefix. If the segment is empty, then no string $\alpha_{i,1}$ has a substring σ_1, and the output is empty; henceforth, we will only consider the case when this segment is non-empty. This segment is transformed into the interval $I_1 = [l_{v_1}, r_{v_1}]$ which indicates the range of their ranks among the ST_1's leaves. Similarly, from σ_2, we obtain the interval I_2. Notice that I_1 is a subinterval of A_1, whereas I_2 is a subinterval of A_2. A given color may occur many times in I_1 if σ_1 occurs many times in $\alpha_{i,1}$ for some i (likewise for $\sigma_2, \alpha_{i,2}$'s and I_2). Finding the distinct colors that occur in *both* I_1 and I_2 is the same problem as finding the distinct string-ids that contain σ_1 and σ_2 in their first and second components, respectively. That completes the reduction to the CCQ problem.

Theorem 4.1. *There exists an algorithm to solve the* 2*-d substring indexing problem using*

$$O\left(\frac{\sum_i \ell_{i,1}\ell_{i,2}}{B}\log_\chi(M)\right)$$

disk pages, where $M = \sum_i (\ell_{i,1} + \ell_{i,2})$, *and* χ *is an integral parameter. Each query takes*

$$O\left(\frac{|\sigma_1| + |\sigma_2|}{B} + \chi \log_\chi(M)\left[\log_B(M) + \frac{\texttt{output}}{B}\log_m\left(\frac{\chi \log_\chi(M)\texttt{output}}{B}\right)\right]\right)$$

page accesses in the worst case, where m is the ratio of the main memory size to the disk page size.

Proof. The reduction of the 2-d substring indexing problem to the CCQ problem goes through the construction of the matrix AA of size $M \times M$, since M is the overall length of the array A. As we observed above, the peculiarity of the current CCQ problem is that the query interval I_1 lies in the first half of array A (i.e. subarray A_1) whereas the query interval I_2 lies in the second half of A (i.e. subarray A_2). Hence, by carefully building the instance of the CCQ problem, we can save some further space. We therefore construct a point colored c for every pair of A's entries such that the first entry lies in A_1 and the second entry lies in A_2. Since we have $\ell_{i,1}$ points (string suffixes) colored i in A_1 and $\ell_{i,2}$ points (string suffixes) colored i in A_2, we may conclude that the number of pairs colored i in AA is $\ell_{i,1}\ell_{i,2}$. Consequently, the number of non-zero entries of AA is actually $T = \sum_i \ell_{i,1}\ell_{i,2}$. By setting $N = M$ in Theorem 3.2, we derive the space bound stated in the theorem.

From [5], we know that the range search for the substring σ_1 in ST_1 and for the substring σ_2 in ST_2 takes $O(\frac{|\sigma_1|+|\sigma_2|}{B} + \log_B(M))$ disk accesses overall, since M is the number of indexed suffixes. Following that, we solve the CCQ problem via the data structure of Theorem 3.2. By setting $N = M$, we derive the query bound stated in the theorem. □

5. Tunable algorithms

In this section, we present a family of algorithms that combine the previously presented CRQ and CCQ solutions to solve the CCQ problem in a hybrid fashion. Our solutions rely on space partitioning. We present it first for the CCQ problem and then apply the technique to the 2-d substring indexing problem.

5.1. Family of CCQ algorithms

5.1.1. Data structure

We partition the segment $[1, N]$ into N^{1-y} intervals of size N^y each ($y < 1$ is a positive constant). Let us denote the set of these intervals using $\mathcal{I}$.

For each interval $I \in \mathcal{I}$ and each color c, we (pre)compute the values $\max\{I, c\}$ and $\min\{I, c\}$ corresponding to the maximum and minimum integer in I colored c (if any), respectively. Then we construct the set P of 2-d points (u, v) such that u is an integer in [1, N] and v is equal to either $\min\{I, c\}$ or $\max\{I, c\}$, for any interval I in $\mathcal{I}$ and for color c of u. Altogether, the set P has size $O(N^{2-y})$ since u can be chosen in N ways and v can be chosen in $O(N^{1-y})$ ways (note that the color c of u is fixed, once u is given).

On set P, we perform all the preprocessing needed for the CCQ algorithm described earlier, in Section 3.

5.1.2. Query processing

Consider answering the original query on intervals I_1 and I_2. We distinguish two cases:

1. Intervals I_1 and I_2 are each entirely contained in some interval of $\mathscr{I}$.
2. At least one of I_1 or I_2 overlaps the boundary of some interval in $\mathscr{I}$.

We answer the query in case (1) by scanning I_1 and I_2, sorting based on colors, and intersecting the two sets. Say we have sorted each of the complete intervals a priori based on colors during a preprocessing step (the cost for this will be dominated by other costs described below). Case (1) then performs $O(\frac{N^y}{B})$ disk accesses for each query.

To answer the query in case (2), we use the CCQ algorithm from Section 3 as follows. Assume that I_1 overlaps the right boundary of the interval I in $\mathscr{I}$ and the left boundary of the interval I' in $\mathscr{I}$ adjacent to I. The crucial observation is the following:

Observation 5.1. *If a color c occurs in I_1, then either* $\max\{I, c\}$ *occurs inside I_1 or* $\min\{I', c\}$ *occurs inside I_1.*

Therefore, if the color c occurs also in I_2 at position v, then the set P contains either the point $(v, \max\{I, c\})$ or the point $(v, \min\{I', c\})$, or both (if c occurs in both I and I'). In any of these three cases, at least one point colored c surely lies inside the rectangle $I_2 \times I_1$. A rectangle query $I_2 \times I_1$ on set P for the CCQ problem will retrieve this color. Clearly, only the colors occurring in both I_1 and I_2 will be retrieved. Using our previous solution, this query performs $O(\chi \log_\chi(N)[\log_B(N) + \frac{\mathtt{output}}{B} \log_m(\frac{\chi \log_\chi(N)\mathtt{output}}{B})])$ page accesses in the worst case, but only $O(\frac{N^{2-y}}{B} \log_\chi(N))$ disk pages are used.

Combining the two cases, and choosing $\chi = 2$, we get the following result:[5]

Theorem 5.1. *The CCQ problem can be solved using $\tilde{O}(\frac{N^{2-y}}{B})$ disk pages, for a positive fraction y. Any query takes*

$$\tilde{O}\left(\frac{N^y + \mathtt{output}}{B}\right)$$

disk page accesses in the worst case.

This result gives a trade-off between preprocessed space and query time, by choosing different y's.

5.1.3. An average-case sweet spot

The above solution actually shows a significantly better average case behavior. Assume that each interval is equally likely to be I_1 (or I_2, independently). The probability that I_1 or I_2 has both endpoints within an interval in $\mathscr{I}$ is N^{2y-2}. Hence, each query performs $O(N^y/B)$ disk page

[5] The notation $\tilde{O}$ suppresses poly-logarithmic terms in the function in general; in this paper, all poly-logarithmic terms are at most quadratic.

accesses with probability N^{2y-2} and $\tilde{O}(\log_B(N) + \frac{\mathtt{output}}{B})$ otherwise. Picking $y = \frac{2}{3}$, we have the "sweet spot" result:

Theorem 5.2. *The CCQ problem can be solved using* $\tilde{O}(\frac{N^{4/3}}{B})$ *disk pages. Each query takes only*

$$\tilde{O}\left(\log_B(N) + \frac{\mathtt{output}}{B}\right)$$

disk accesses on average.

5.2. Tunable solution for 2-d substring indexing problem

We solve the 2-d substring indexing problem by reducing it to the CCQ problem as before. However, we choose a different parameterization from the one in the theorem above. We divide the A array into $\sqrt{L}$ intervals of equal size, where L is the average length of the $\alpha_{i,j}$'s.

Consider case (1) as before. We solve it differently: instead of merging the intervals (which would take $\frac{M}{B\sqrt{L}}$ disk accesses), we solve two independent CRQ problems and merge the resulting lists so that the query takes $O(\frac{\mu \log_m(\mu/B)}{B})$ disk accesses in the worst case; here, $\mu(\leqslant M)$ is the maximum of the number of string pairs in which σ_1 appears and in which σ_2 appears, and m is the ratio of main memory size to page size. (This is independent of the size of the interval.)

Case (2) is solved as before. However, the total number of non-zero entries we deal with is only $O(M\sqrt{L})$. Hence, the number of disk pages used is $\tilde{O}(\frac{M\sqrt{L}}{B})$ and each query takes $\tilde{O}(\log_B(M) + \frac{\mathtt{output}}{B})$ disk accesses.

The hybrid algorithm therefore uses the CRQ or CCQ problem according to the suitable case above. The various tradeoffs for the 2-d substring indexing problem follow from the various tradeoffs we have already established for the CCQ problem.

Unlike the CCQ problem, described in the previous subsection, it is difficult to infer an average case behavior for the 2-d substring indexing problem based on this hybrid algorithm. This is because a suitable model of "average" case queries is not apparent. However, we make the following observation: say we are given the probability distribution of each node u in the string B-tree ST_1 representing the longest common prefix between the query $\sigma_{i,1}$, and likewise for nodes in ST_2; furthermore, these probability distributions are independent. Then, we can set up a dynamic programming algorithm and determine a way to partition array A so that the number of times Case (1) is employed is minimized (this case is the bottleneck in disk accesses for query processing).

6. Conclusions

Multi-dimensional substring matching is becoming crucial for supporting the next generation of database applications. Solving this problem efficiently requires good multi-dimensional substring indexing techniques. In this paper, we provide the *first* non-trivial bounds known for the 2-d substring indexing problem: based on a novel geometric approach, we provide I/O efficient algorithms for this problem. This establishes a formal foundation for subsequent work in this

area. Our approach can be extended to d dimensions, but it seems difficult to get a better than $N^{d-1+\varepsilon}$ space bound with sublinear query time. It is an open problem to get substantially improved tradeoffs.

References

[1] P. Agarwal, J. Erickson, Geometric range searching and its relatives, in: B. Chazelle, J. Goodman, R. Pollack (Eds.), Advances in Discrete and Computational Geometry, Amer. Math. Soc., Providence, RI, 1998.
[2] L. Arge, V. Samoladas, J.S. Vitter, On two-dimensional indexability and optimal range search indexing, Proceedings of ACM Principles of Database Systems (PODS), Philadelphia, PA, 1999, pp. 346–357.
[3] R. Bayer, K. Unterauer, Prefix B-trees, ACM Trans. Database Systems 2 (1) (1977) 11–26.
[4] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger, The R*-tree: an efficient and robust access method for points and rectangles, Proceedings of ACM SIGMOD, Atlantic City, NJ, 1990, pp. 220–231.
[5] P. Ferragina, R. Grossi, The string B-tree: a new data structure for string search in external memory and its applications, J. ACM 46 (2) (1999) 237–280.
[6] V. Gaede, O. Günther, Multidimensional access methods, ACM Comput. Surveys 30 (2) (1998) 170–231.
[7] G. Gonnet, R. Baeza-Yates, T. Snider, New indices for text: PAT trees and PAT arrays, in: Information Retrieval: Data Structures and Algorithms, Prentice-Hall, Englewood Cliffs, NJ, 1992.
[8] P. Gupta, R. Janardan, M. Smid, Further results on generalized intersection searching problems: counting, reporting, and dynamization, J. Algorithms 19 (1995) 282–317.
[9] A. Guttman, R-trees: a dynamic index structure for spatial searching, Proceedings of ACM SIGMOD, Boston, MA, 1984, pp. 47–57.
[10] H.V. Jagadish, N. Koudas, D. Srivastava, On effective multi-dimensional indexing for strings, Proceedings of ACM SIGMOD, Dallas, TX, 2000, pp. 403–414.
[11] U. Manber, G. Myers, Suffix arrays: a new method for online string searches, SIAM J. Comput. 22 (5) (1993) 935–948.
[12] Y. Matias, S. Muthukrishnan, S. Sahinalp, J. Ziv, Augmenting suffix trees with applications, Proceedings of European Symposium on Algorithms, Venice, Italy, 1998, pp. 67–78.
[13] E.M. McCreight, A space-economical suffix tree construction algorithm, J. ACM 23 (1976) 262–272.
[14] D.R. Morrison, PATRICIA: practical algorithm to retrieve information coded in alphanumeric, J. ACM 15 (4) (1968) 514–534.
[15] H. Samet, The Design and Analysis of Spatial Data Structures, Addison-Wesley, Reading, MA, 1990.
[16] T. Sellis, N. Roussopoulos, C. Faloutsos, The R+-tree: a dynamic index for multi-dimensional data, Proceedings of VLDB, Brighton, England, 1987, pp. 507–518.
[17] J.S. Vitter, E.A.M. Shriver, Optimal disk I/O with parallel block transfer, Proceedings of Symposium on Theory of Computing (STOC), Baltimore, MD, 1990, pp. 159–169.